OBJECT DELINEATION IN SATELLITE IMAGES

Zhuocheng Shang
Computer Science and Engineering
University of California, Riverside
zshan011@ucr.edu

Ahmed Eldawy
Computer Science and Engineering
University of California, Riverside
eldawy@ucr.edu

ABSTRACT

Machine learning is being widely applied to analyze satellite data with problems such as classification and feature detection. Unlike traditional image processing algorithms, geospatial applications need to convert the detected objects from a raster form to a geospatial vector form to further analyze it. This gem delivers a simple and light-weight algorithm for delineating the pixels that are marked by ML algorithms to extract geospatial objects from satellite images. The proposed algorithm is exact and users can further apply simplification and approximation based on the application needs.

1 Introduction

There has been a recent increase in machine learning algorithms and applications that operate on high-resolution satellite data such as land use classification and object detection [1, 2]. This increase has been driven by the public availability of satellite data and the recent advancements in machine learning. Many of these algorithms, such as object detection, produce their output by marking pixels on the satellite data. For regular image processing, this output can be enough. However, for geospatial applications, it is desirable to delineate pixels that correspond to one object to form a geospatial polygon that can be further processed in GIS applications.

This gem introduces a light-weight algorithm that delineates marked pixels on a satellite image to produce a valid geospatial polygon, i.e., closed and not self-intersecting. The proposed algorithm is exact in the sense that it exactly delineates all marked pixels with no approximation. Based on its needs, an application can further apply simplification algorithms to produce the desired output.

In the field of graphics, *image tracing* algorithms are widely used to vectorize raster images [3, 4]. The goal of these algorithms is usually to produce basic geometric shapes, e.g., circles and lines, which might entail simplification that is not always desirable for geospatial applications. The proposed algorithm is based on the idea of image tracing but is tailored for geospatial data.

Figure 1 gives an overview of the object delineation problem. The gray pixels are the ones marked by the ML algorithm and our goal is to produce the polygon marked by the arrows. To provide an exact answer on satellite data, the output polygons must consist only of orthogonal lines. The key idea is to scan the image once with a 2×2 window to find all the polygon vertices and connect them in the correct order as shown on the figure.

2 Extracting Objects

This section describes two steps in the proposed approach that delineates objects from satellite data, *orthogonal lines detection* and *ring formation*. The detection step locates all orthogonal lines from occupied pixels in a single scan over the image. The ring formation step combines orthogonal lines into geospatial linear rings.

Figure 1 gives an overview of what the algorithm does. Given a raster image with marked pixels, it creates an orthogonal polygon that surrounds all marked pixels. The vertices that make polygons are all located at pixel corners as shown in figure. By convention, the vertices of a polygon is ordered in counter clock-wise order (CCW). In case of a polygon with a hole, the vertices of the hole are ordered in clock-wise order (CW).

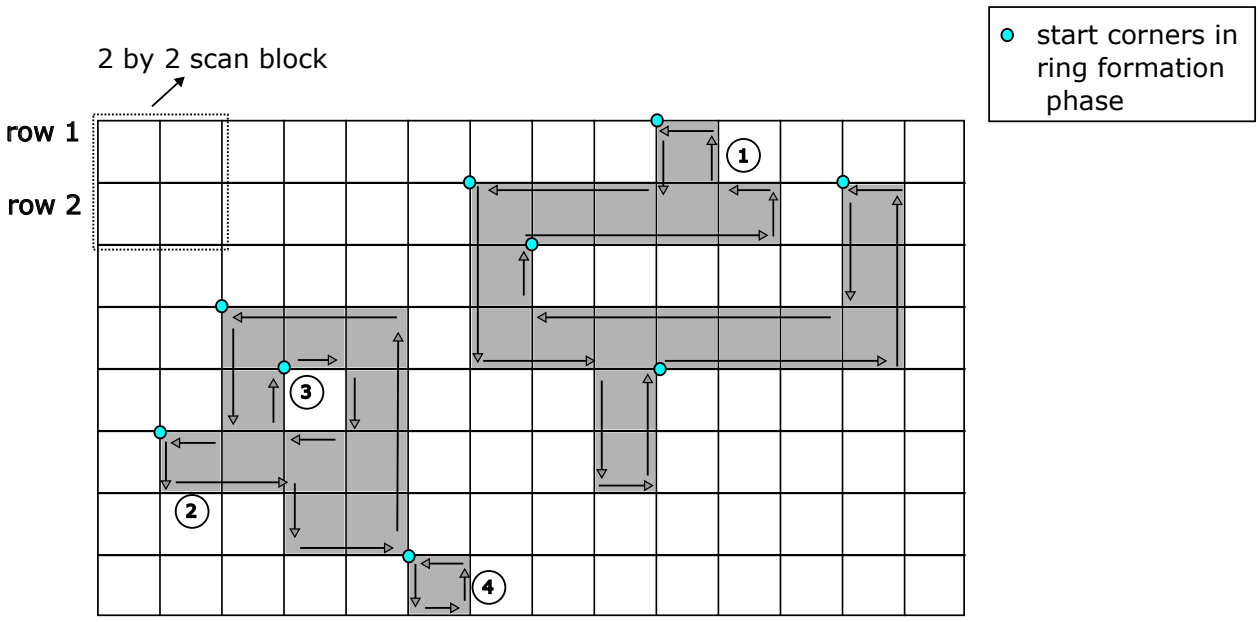


Figure 1: Example of occupied pixels with orthogonal lines

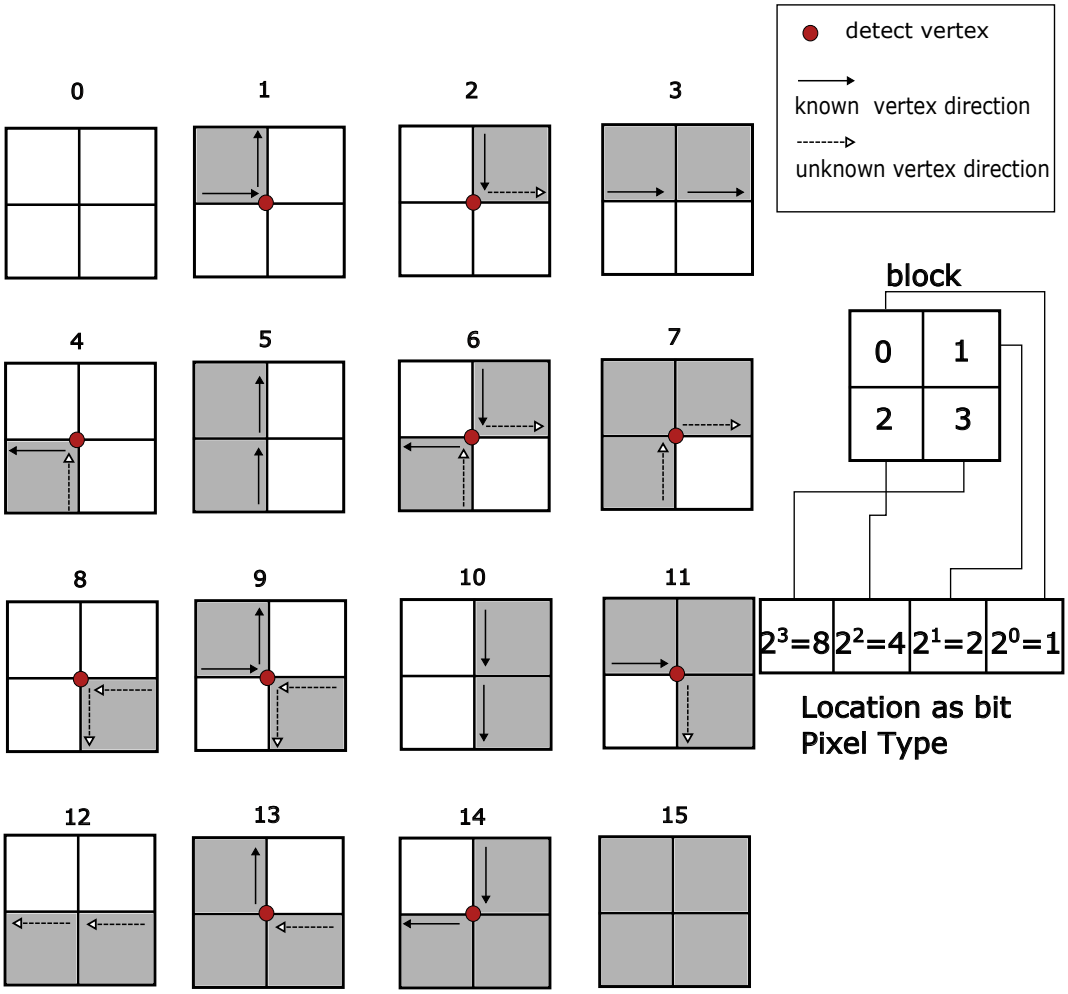


Figure 2: Different pixel occupied cases within one 2 by 2 block

2.1 Orthogonal Lines Detection

This step takes the marked raster as input and produces all the orthogonal lines that comprise all the polygons. The lines are grouped in rings, i.e., a circular linked list of vertices, as shown in Figure 1. The key observation is that each vertex on the polygon connects a horizontal edge to a vertical edge. Thus, to find all vertices, we need to locate the parts of the image where a horizontal edge meets a vertical edge. Then, we should connect these vertices in the correct order to create a closed polygon. We can observe that a vertex can be detected by checking the 2×2 window where the vertex is at the center. This is enough to detect that a horizontal and vertical edges meet. Figure 1 illustrates an example 2×2 window. This step simply runs a sliding window over the entire raster to create all the vertices and arrange them in the correct order as further detailed below. The sliding window starts from the top-left and slides over each row from left to right.

Since a 2×2 window contains only four pixels, and each pixel can either be marked or unmarked, there is a total of 16 possible cases that can happen. If we handle all of them correctly, then we know that we have all possible cases covered. Figure 2 illustrates all the 16 cases. For formalization, we identify these cases by assigning a bit position to each of the four pixels as show in the figure. We can immediately see that cases 0, 3, 5, 10, 12, and 15 do not result in any detected vertices. Cases 1, 2, 4, 7, 8, 11, 13, and 14, each result in creating one vertex at the center of the window. Cases 6 and 9 are special cases that result in two coinciding vertices, both at the center. We chose to create these two vertices to ensure that we create non-intersecting and non-overlapping polygons. If our goal is to only find the location of the vertices without caring about their connection and order, then it is enough to scan all 2×2 windows and emit a vertex for each of the above cases. However, we also want to connect them in the correct order which we describe below.

We notice that each vertex must connect a horizontal edge to a vertical edge. The horizontal edge must be at the same row and the vertical edge must be at the same column. Following our convention, the direction of the created edges is shown in Figure 2. However, we notice that we cannot always create the edge when we create a vertex. For example, Case 1 creates a vertex that connects another vertex to the left to a vertex to the top. This is an easy case because both vertices must have been already created following our sliding window order, i.e., left-to-right and top-to-bottom. However, Case 2 is harder since it connects a vertex to the top to a vertex to the right that is not yet created. Hence, while handling Case 2, we can only create the vertical edge and not the horizontal edge. In Case 7, we cannot create any edge since the end points of both edges are not created yet. In Figure 2, solid arrows indicate the edges that can be created in each case while dotted arrows indicate edges that cannot be created at that point.

To resolve the issue of incomplete edges, we define the notion of an *open vertex*. An open vertex is part of an edge that is not yet created, i.e., the other end point is not detected yet. Since all edges are orthogonal, each open vertex can only be paired with another vertex at the same row or the same column. In addition, given the order at which the 2×2 window slides, a vertical edge can only have an open vertex at the top and a horizontal edge can only have an open vertex to the left. Therefore, as we scan, we keep at most one open vertex to the left and at most $w + 1$ open vertices at the top, one for each column, assuming the raster data has a width of w pixels.

Given these open vertices, all edges can be created efficiently in one scan. If a vertex is the top vertex of a vertical edge or a left vertex of a horizontal edge, it is stored as an open vertex. On the other hand, if it is the bottom vertex of a vertical edge or a right vertex of a horizontal edge, it is paired with the corresponding open vertex.

In Figure 2, Case 1 creates a vertex that connects the left vertex to the top vertex. Both left and top vertices must be open vertices that have been created before reaching that case but without knowing the other end point. Thus, we will update the next link of the left open vertex to point to the newly created vertex and update the next link of the new vertex to point to the top vertex. On the other hand, Case 2 connects the top vertex to a vertex to the right that is not reached yet. Therefore, we updated the next link of the vertex at the top to point to the new vertex. Then, we keep this new vertex as an open vertex to the left to be paired later. In Case 7, the newly created vertex is both an open left vertex and an open top vertex so it will be stored as that.

Cases 6 and 9 are more interesting. It indicates two pixels meeting at a corner. To keep the created polygons valid, we create two coinciding vertices at the center. In other words, Case 6 does the work of Cases 2&4 together and Case 9 combines Cases 1&8 together.

Algorithm 1 gives the pseudo-code of detecting orthogonal lines. The input is a two-dimensional bit array of width w and height h. The output is a set of circular linked lists of vertices. Each vertex has an integer coordinate (x, y), a pointer to the *next* vertex, and Boolean *visited* flag that will be used in the next algorithm. The output is stored in a list of *corners* that contains at least one pointer for each linked list.

We keep a list of open vertices at the top and a single pointer to the left open vertex. All these are initialized to null. We run a loop over all pixels that slides a window at the center of each intersection (x, y) on the raster grid. Then, it computes the pixel type $[0, 15]$ by inspecting the four pixels in the window. Any pixel that falls outside the raster grid is

assumed to be non-marked. After that, it runs a single switch statement that efficiently handles all the cases. To keep track of all linked lists, we need to store at least one pointer in each circular linked list. We chose to record the top-left corner only which is handled by Cases 7,8, and 9.

To analyze the time complexity of this algorithm, we note that the cost is mainly in the for loop that iterates over each pixel. The switch statement has a constant-time cost. Thus, the time complexity is $O(w \cdot h)$ which is linear in terms of number of pixels. For space complexity, we observe that, in addition to the input, we need to keep track of open vertices at the top which requires $O(w)$ space. Also, we need to keep track of all the vertices which requires $O(|V|)$. So, the space complexity is $O(|V| + w)$ which is output-sensitive.

Algorithm 1: Orthogonal Lines Detection

Input : R: Two-dimensional bit array $[w][h]$ for marked pixels
Output : Linked List of vertices $\langle x : Int, y : Int, next : Vertex, visited = false \rangle$

```
1  topVertices: Array of open vertices at each column of size w + 1
2  leftVertex: The open vertex to the left or null
3  Corners: List of start corners
4  for y ∈ [0, h], x ∈ [0, w] do
5      block 0 ← 1 if R[x-1][y-1] not empty, otherwise 0
6      block 1 ← 2 if R[x][y-1] not empty, otherwise 0
7      block 2 ← 4 if R[x-1][y] not empty, otherwise 0
8      block 3 ← 8 if R[x][y] not empty, otherwise 0
9      pixelType ← (block 0 + block 1 + block 2 + block 3)
10     switch pixelType do
11         case 0,3,5,10,12,15 do nothing
12         case 1 do leftVertex.next ← V⟨x, y, topVertices(x)⟩
13         case 2 do leftVertex ← topVertices(x).next ← V⟨x, y, null⟩
14         case 4 do topVertices(x) ← V⟨x, y, leftVertex⟩
15         case 6 do
16             v1 ← topVertices(x).next ← V⟨x, y, null⟩
17             topVertices(x) ← V⟨x, y, leftVertex⟩
18             leftVertex ← v1
19         end
20         case 7,8 do Corners ≪ topVertices(x) ← leftVertex ← V⟨x, y, null⟩
21         case 9 do
22             leftVertex.next ← V⟨x, y, topVertices(x)⟩
23             Corners ≪ topVertices(x) ← leftVertex ← V⟨x, y, null⟩
24         end
25         case 11 do topVertices(x) ← leftVertex.next ← V⟨x, y, null⟩
26         case 13 do leftVertex ← V⟨x, y, topVertices(x)⟩
27         case 14 do topVertices(x).next ← V⟨x, y, leftVertex⟩
28     end
29 end
```

2.2 Ring Formation

This step takes as input the circular linked lists created by the first step and combines each one into a single ring. Based on how the rings were formed, outer rings and inner holes are ordered in CCW and CW order, respectively. To complete merging in one round, the implemented algorithm activates by picking one start corner from a preserved list, shown as dots in Figure 1, and then goes over the entire list. Each vertex in the list is converted to a geospatial coordinate $(longitude, latitude)$ and they are then combined to produce the geospatial ring. To convert integer raster coordinates (x, y) to geospatial coordinates $(longitude, latitude)$, we use an affine transformation, termed grid-to-world. This is a standard method to encode geospatial coordinates of raster datasets. While iterating over the vertices, they are marked as visited by setting the flag in each vertex. This ensures that each ring is converted only once since one ring can contain multiple corners, e.g., ring 1 in Figure 1. After one ring is formed, it is appended to a list of rings which are then returned by the algorithm.

Algorithm 2 provides the pseudo-code of the ring formation process. The input is the list of corners created by the first step. It loops over all corners that are not yet visited. For each corner, it follows the linked list until it goes back

to the start since it is a circular linked list. It converts each vertex to geospatial coordinates using the grid-to-world transformation. Finally, it appends the first point again to close the ring and appends it to the list of rings.

To analyze the time complexity of this algorithm, we observe that the major part is going through all vertices stored in all the linked lists. The time complexity is linear in term of number of vertices stored, which is $O(|V|)$. Thus, the time complexity is output-sensitive. Both the input and output sizes are equal to number of vertices so the space complexity is also $O(|V|)$.

Algorithm 2: Ring Formation

```
Input  : Corners: List of start Corners;
Output : Rings: List of Geometry Linear Rings
for corner ← Corners do
    if not corner.visited then
        p ← corner;
        Coordinates← ⟨⟩: List of geospatial coordinates
        do
            (longitude,latitude) ← grid_to_world(p.x, p.y)
            Coordinates ≪ (longitude, latitude)
            p.visited ← true
            p ← p.next
        while p != corner;
        (longitude,latitude) ← grid_to_world(p.x, p.y)
        Coordinates ≪ (longitude, latitude)
        Rings ≪ Coordinates
    end
    return Rings
end
```

3 Experimental Result

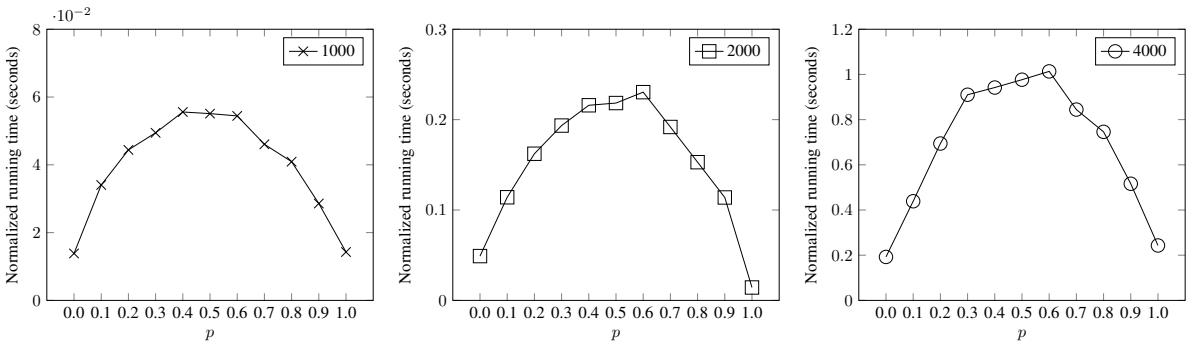
In this part, we run some basic experiments to confirm the scalability of the proposed algorithm. We generate random marked rasters of resolutions 1000×1000 up-to 4000×4000 as shown in Table 1. For each raster size, we mark each pixel with an independent Bernoulli distribution with parameter p. In other words, we scan over all the pixels and randomly mark each pixel with a probability $p \in [0, 1]$. The higher the value of p, the more pixels will be marked in the raster. In this experiment, we vary p from 0.0 to 1.0 in increments of 0.1. For each value of p, we generate 100 random rasters and compute the average running time.

Figure 3 shows the average running time of the proposed algorithm. In general, as expected, the entire running time of the whole algorithm linearly increases as enlarge the raster size. Notice that the number of pixels increases by a factor of four in each subfigure and so as the peak running time.

In each figure, the running time shows a bitonic behavior where it starts very small, peaks around the range $[0.4, 0.6]$ and then starts to fall down again. This can be explained by the output-sensitivity of the algorithm. For both very small and very large values of p, there are only a very few vertices to be detected since the entropy is low. Thus, the ring formation step of the algorithm finishes very quickly since it will have a few vertices to trace. When the entropy peaks at 0.5, the algorithm takes the longest running time since it will detect the largest number of vertices. This behavior confirms our analysis of output-sensitivity. In reality, when there are real objects to be detected in an image, the entropy will be low and hence the algorithm will run much faster than the peak performance in the figure. The entropy is the highest when the image is purely random which is not expected in real scenarios.

Table 1: Experimental test cases setup

Raster W×H	Probability of marked pixel (p)
1000×1000	[0.0 , 1.0]
2000×2000	[0.0 , 1.0]
4000×4000	[0.0 , 1.0]



(a) Raster size 1000 × 1000 (b) Raster size 2000 × 2000 (c) Raster size 4000 × 4000

Figure 3: Running time with different raster and occupied pixels size

References

[1] Anju Asokan and J Anitha. Machine learning based image processing techniques for satellite image analysis -a survey. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 119–124, 2019.

[2] Gong Cheng and Junwei Han. A survey on object detection in optical remote sensing images. *ISPRS Journal of Photogrammetry and Remote sensing*, 117:11–28, 2016.

[3] Peter Selinger. Potrace: a polygon-based tracing algorithm. *Potrace (online), http://potrace. sourceforge. net/potrace. pdf (2009-07-01)*, 2, 2003.

[4] Edoardo Alberto Dominici, Nico Schertler, Jonathan Griffin, Shayan Hoshyari, Leonid Sigal, and Alla Sheffer. Polyfit: Perception-aligned vectorization of raster clip-art via intermediate polygonal fitting. 39(4), jul 2020.